DESENVOLVIMENTO DE SOFTWARE

Um pouco sobre arquitetura.

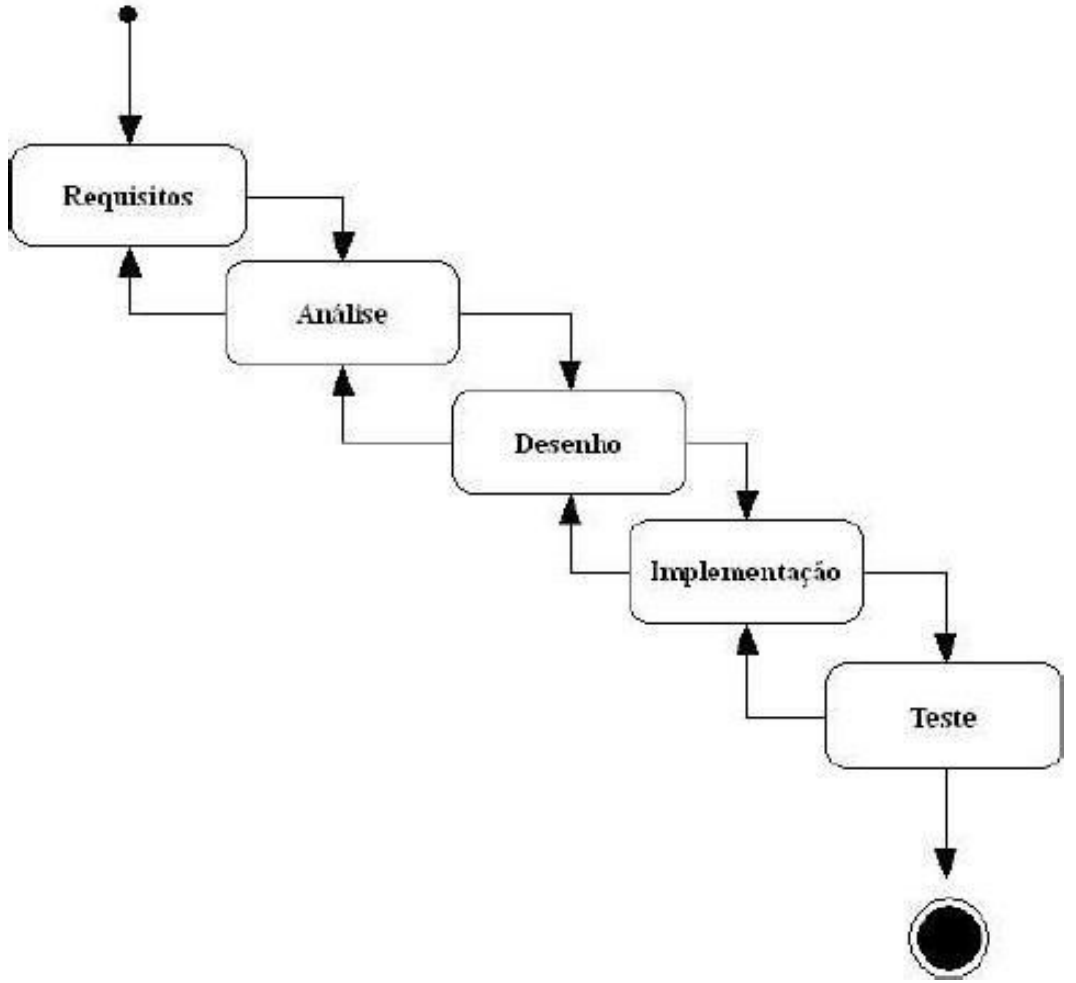
Fases do ciclo de vida

- 1. Levantamento de Requisitos
- 2. Análise e Projeto (Design)
- 3. Implementação (Desenvolvimento)
- 4. Testes
5. Implantação (Deploy)
6. Manutenção
7. Evolução

Modelos de ciclo de vida

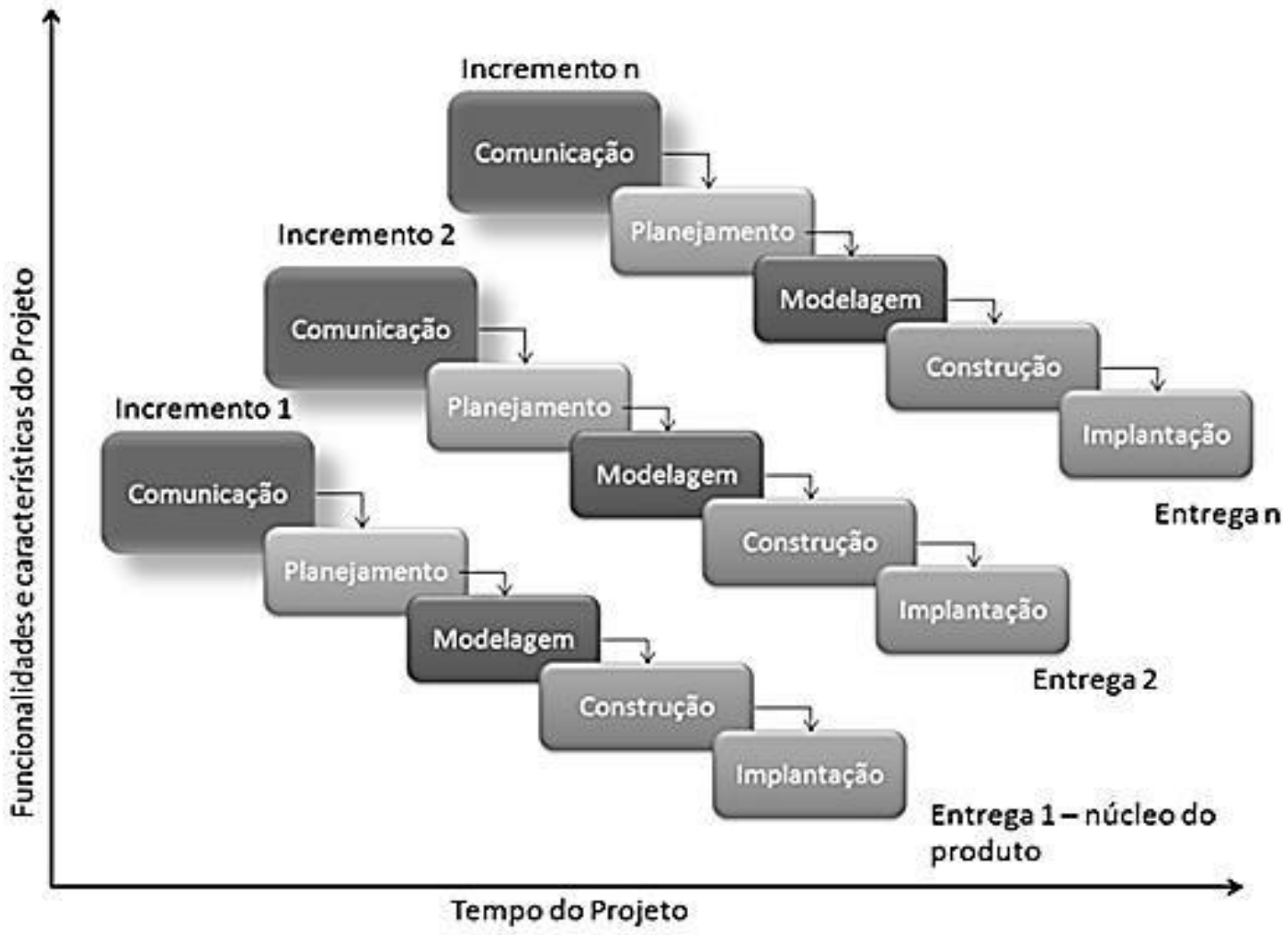
Modelo Cascata

- O desenvolvimento segue uma sequência linear de fases.
- **VANTAGENS**: Simples de entender e fácil de gerenciar em projetos com requisitos bem definidos.
- **DESVANTAGENS**: Rigidez para mudanças. Se um erro for detectado tardiamente, é caro corrigir.



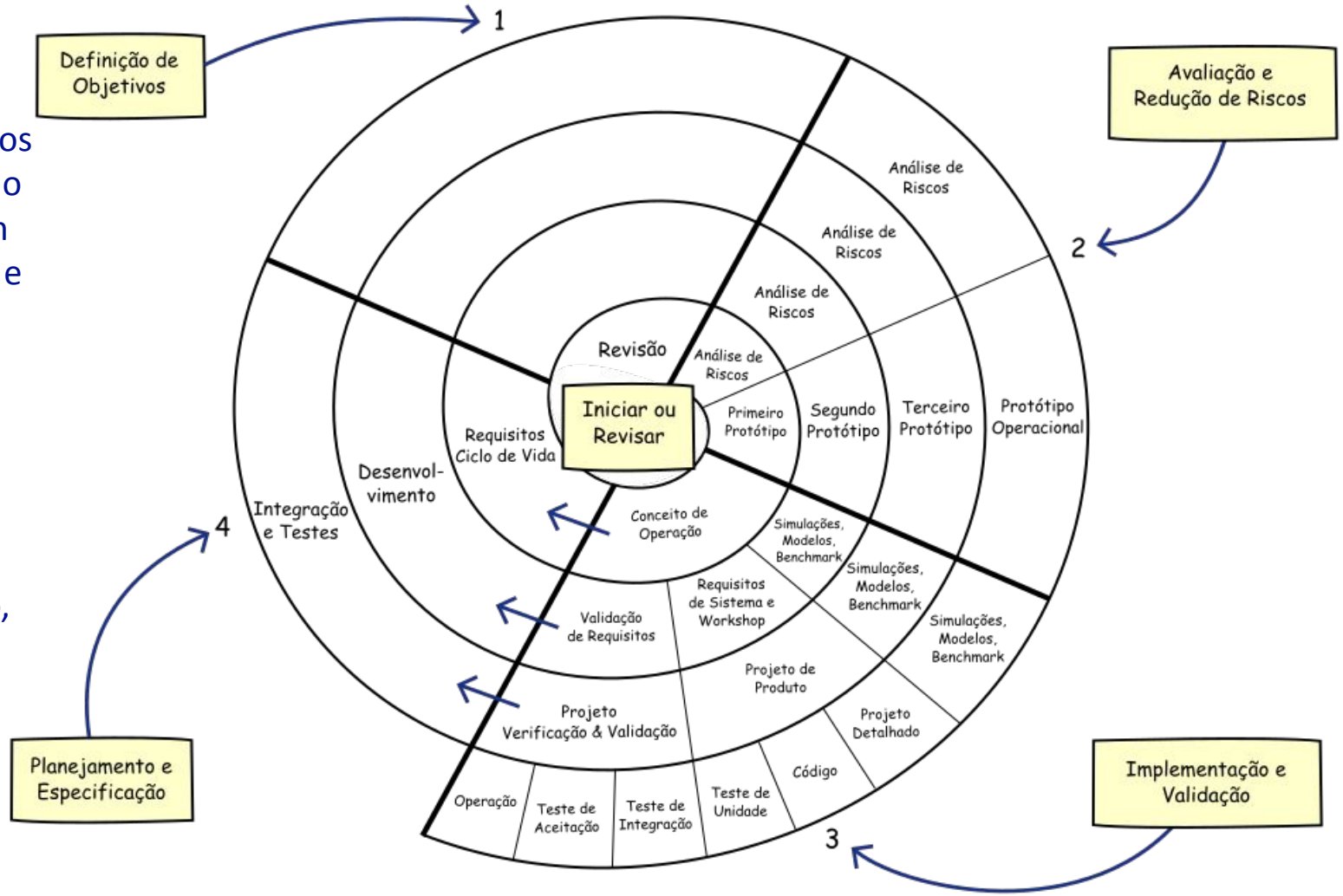
Modelo Incremental

- O desenvolvimento é feito em partes, ou incrementos, onde cada versão adiciona funcionalidades ao produto.
- **VANTAGENS**: Funcionalidades são entregues de forma contínua, permitindo feedback mais rápido.
- **DESVANTAGENS**: Requer planejamento cuidadoso para integrar incrementos sem criar inconsistências.



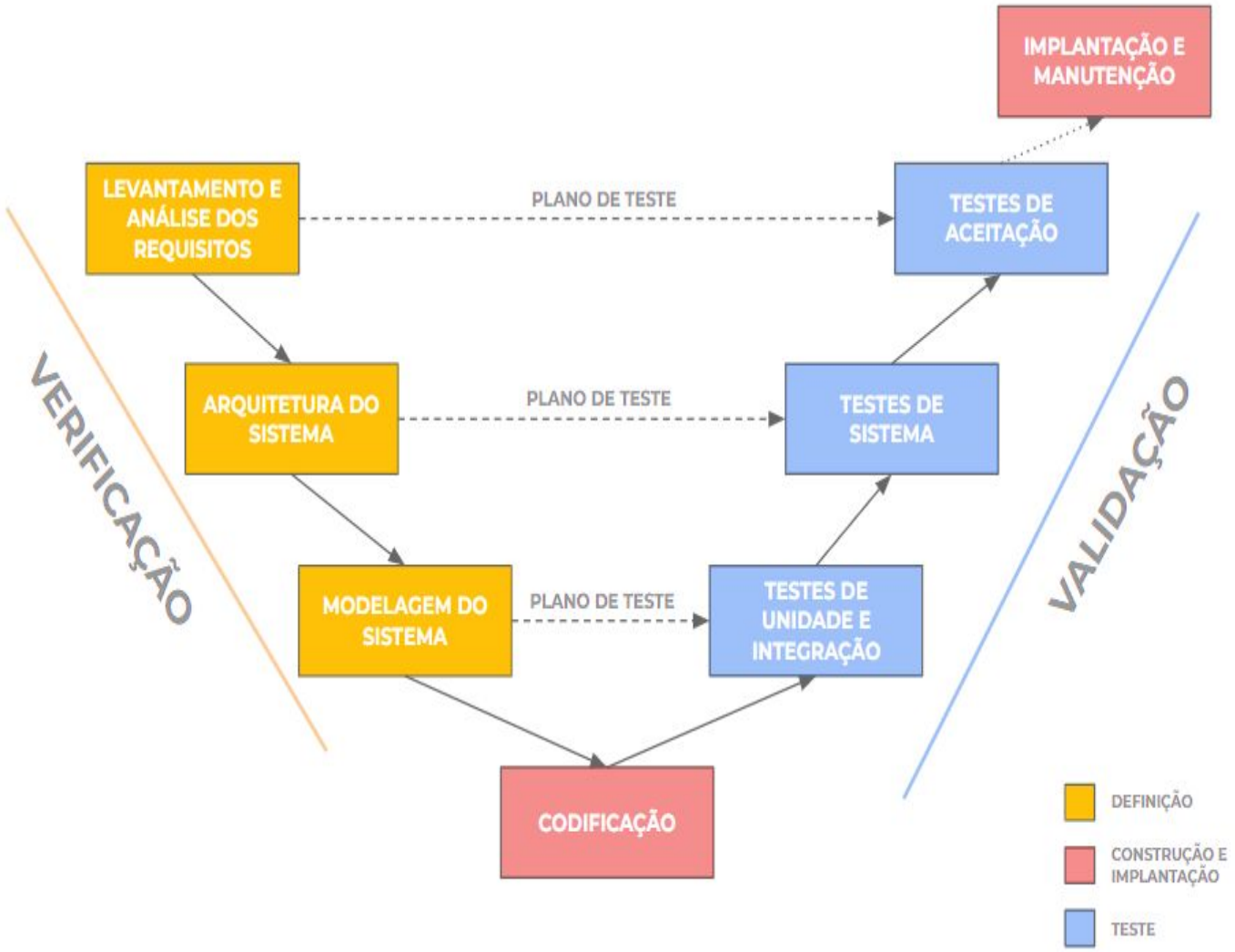
Modelo Espiral

- O desenvolvimento ocorre em ciclos repetitivos (iterações), combinando elementos do modelo cascata com uma abordagem de prototipagem e análise de riscos.
- **VANTAGENS**: Permite a gestão de riscos e revisões frequentes.
- **DESVANTAGENS**: Complexo e caro, requer uma equipe experiente em análise de riscos.



Modelo V

- Similar ao modelo cascata, mas cada fase tem uma atividade de validação correspondente (por exemplo, os testes de validação estão diretamente conectados à fase de design).
- **VANTAGENS**: Forte foco em verificação e validação em cada etapa do desenvolvimento.
- **DESVANTAGENS**: É rígido e difícil de adaptar a mudanças.



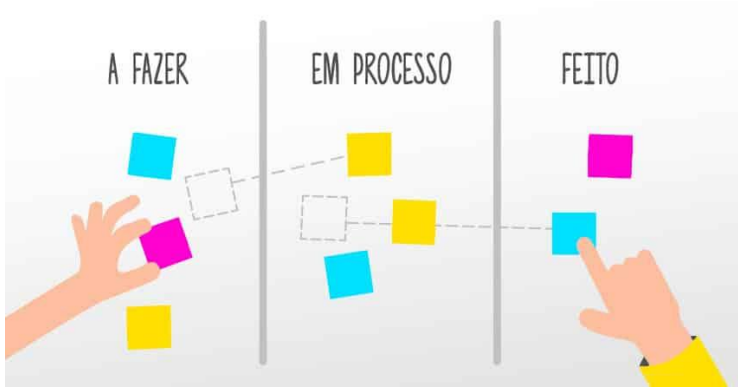
Metodologias Ágeis

✔ Scrum

- **Fases:** Dividido em sprints curtos (2 a 4 semanas), ao final de cada sprint é **entregue** um incremento funcional do produto.
- **Características:** Reuniões diárias, forte colaboração e feedback constante do cliente.

✔ Kanban

- **Fases:** Fluxo contínuo de trabalho, visualizado através de um quadro de tarefas que passam por várias fases (como "A fazer", "Em andamento", "Concluído").
- **Características:** Foco na eficiência e na limitação do número de tarefas do trabalho em progresso.



Metodologias Ágeis

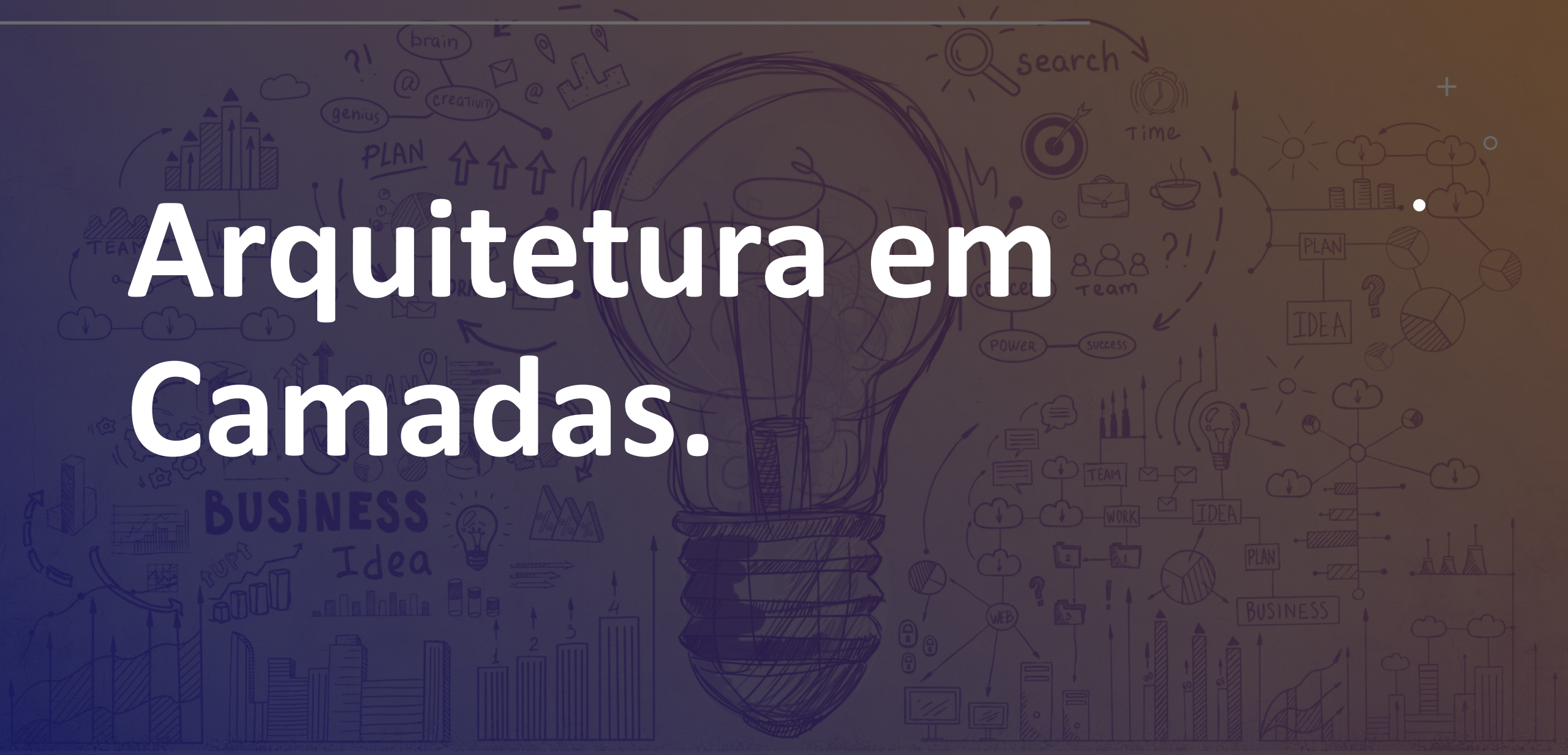
✔ XP - Extreme Programming

- **Fases:** Focadas na melhoria contínua, como desenvolvimento orientado a testes, integração contínua e refatoração frequente.

- **Características:** Qualidade de código e agilidade na resposta a mudanças. Desenvolver os testes antes da solução.

Comunicação – Simplicidade
Feedback – Coragem - Respeito

Arquitetura em Camadas.



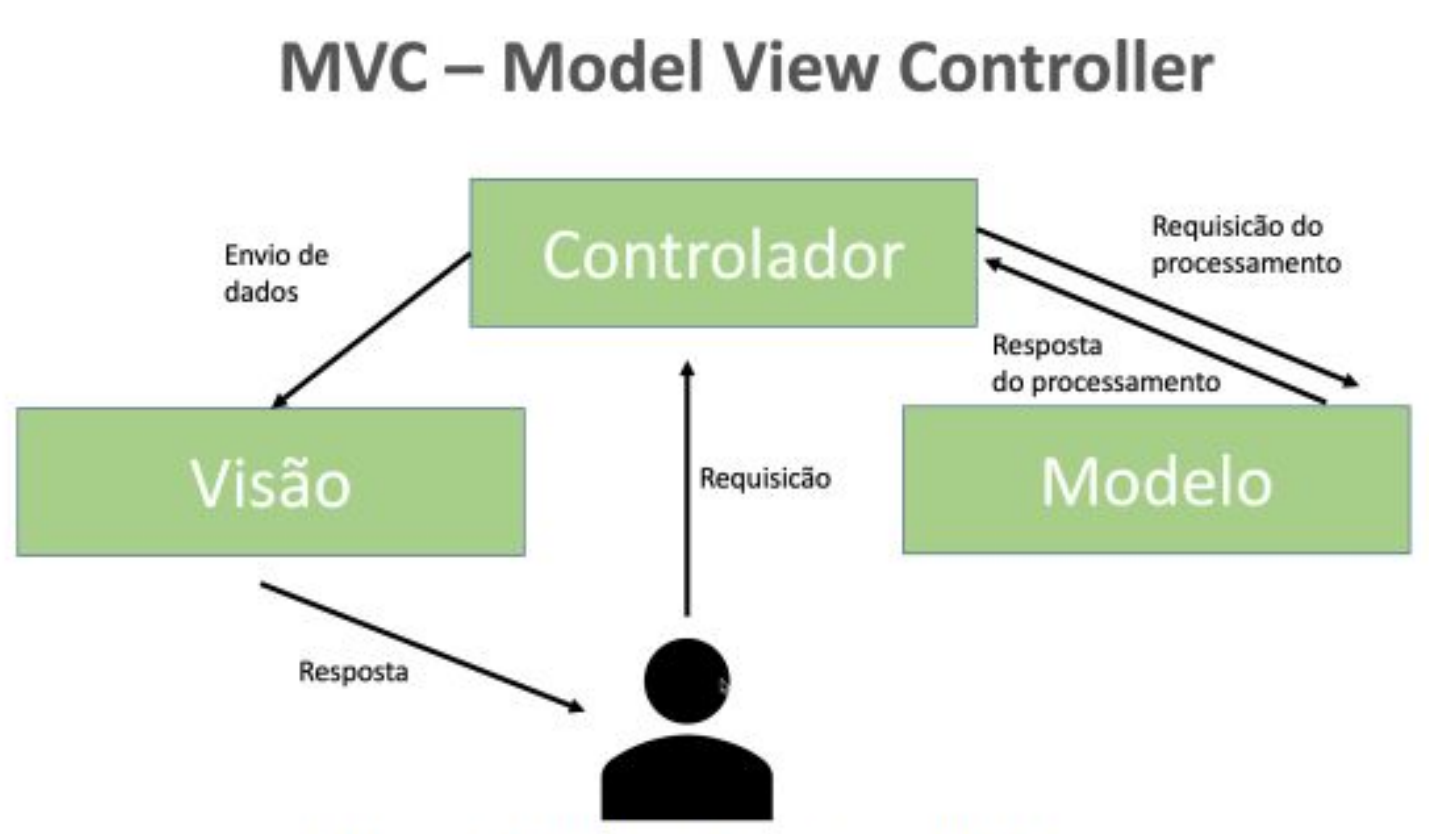
Estilos Arquiteturais em camada

- MVC

Controlador - O Código que lida com a interação do usuário e informa ao Modelo e o View para mudar conforme apropriado.

Visão – Parte da aplicação que representa a apresentação de dados.

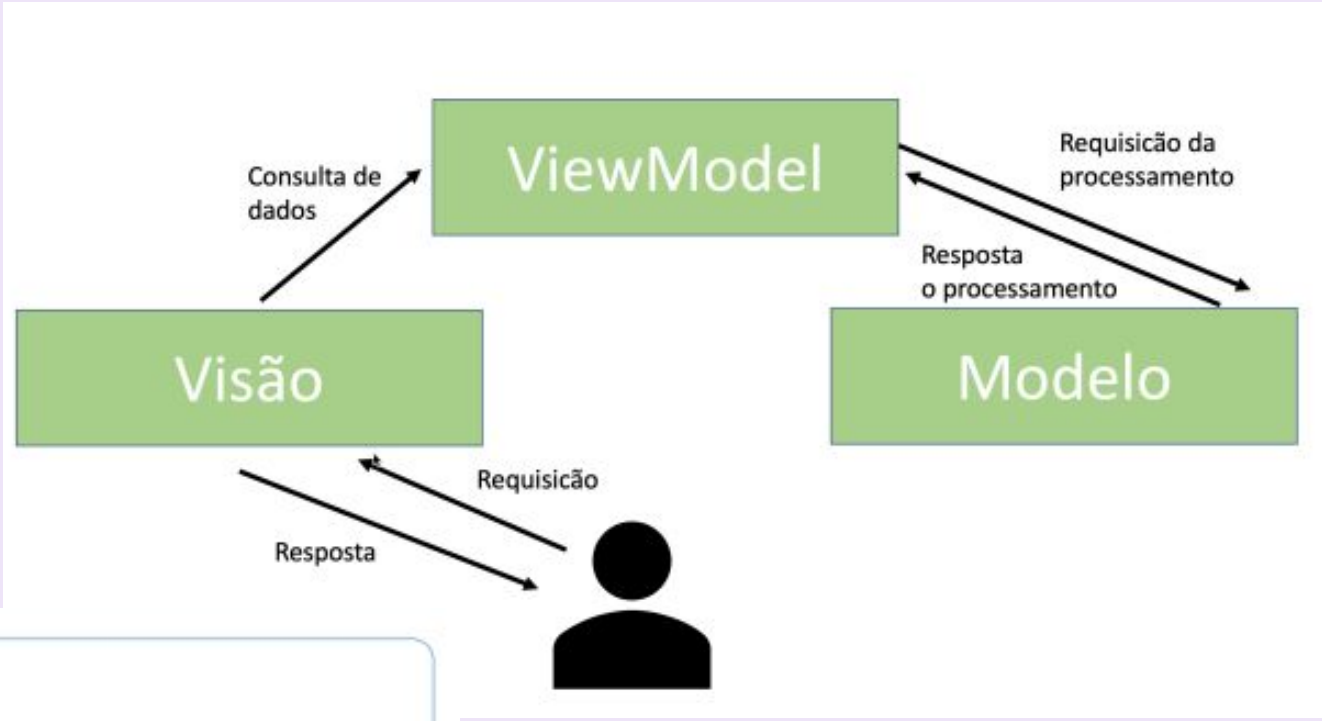
Modelo - Armazena dados e lógica relacionada.

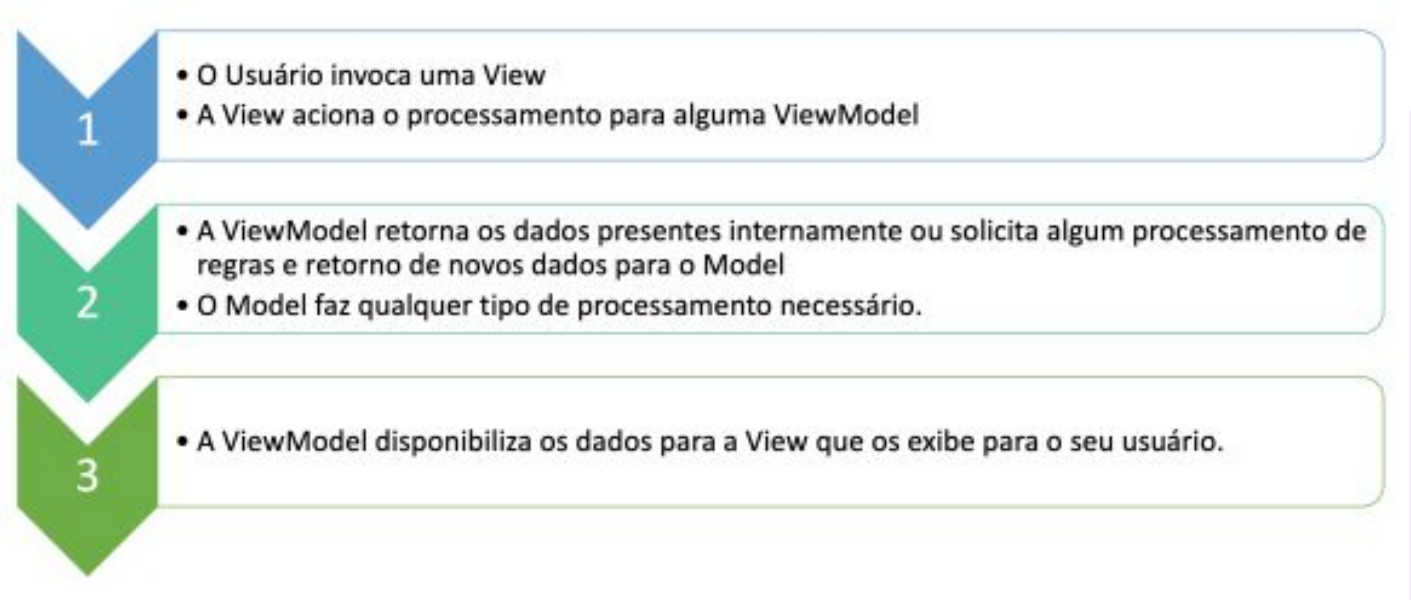


• MVVC

Desenvolvido a partir dos anos 90 para a escrita de aplicações Desktop.
Ele foi popularizado por modelos visuais tais como o Java Swing, Microsoft Silverlight e afins.

Nos últimos anos, com a crescente complexidade das interfaces Web, ele foi adotado em frameworks tais como Angular, React ou Vue.JS.





Modelo - Armazena dados e lógica relacionada.

Visão – Componentes de interface (HTML, CSS)

View Model - responsável por apresentar funções, comandos, métodos, para apoiar o estado do View. (DOM)

DDD – Domain-Driven Design

- ✔ Abordagem para o desenvolvimento de software que
- ✔ Foca na criação de soluções baseadas no entendimento de domínio(problema)
- ✔ A ideia central é criar um modelo que reflita o domínio de negócios e usar esse modelo para guiar o desenvolvimento do software.

Domínio: É o problema. Gerenciamento de estoque, o domínio é "gestão de estoque".

Modelo de Domínio: É uma representação. O modelo é criado com base na colaboração entre desenvolvedores e especialistas do domínio (pessoas que conhecem bem o negócio).

Entidades: São objetos que têm uma identidade própria. Exemplo "Cliente" ou um "Produto"

Valor Objetos: São objetos que são definidos por seus atributos. Por exemplo, um "Endereço“.

Agregados: São um grupo de entidades e valor objetos que são tratados como uma unidade única para garantir a consistência. Por exemplo, um "Pedido" pode ser um agregado que inclui o "Cliente" e os "Itens do Pedido".

Serviços: São operações ou funcionalidades necessárias para o domínio. Por exemplo, um serviço pode calcular o preço total de um pedido.

Fábricas: É uma classe responsável por encapsular a lógica de criação dos objetos, e para garantir que eles sejam criados de forma consistente. Por exemplo, um novo cliente com todos os atributos preenchidos corretamente.

• DDD – Por que usar?

A vantagem mais óbvia do DDD é fazer todos usarem a mesma linguagem. Quando as equipes de desenvolvimento usam a mesma linguagem dos especialistas no domínio, o resultado é um design de software que faz sentido para o usuário final. Como a terminologia da aplicação corresponde às atividades do mundo real, há menos confusão, e os usuários aprendem mais rapidamente a usar o produto.

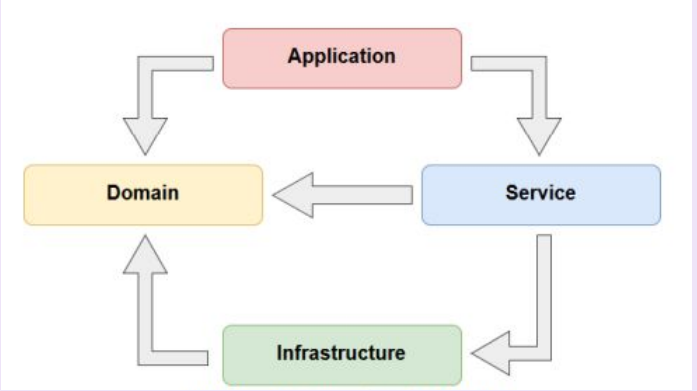
Entidades: Cliente, Produto, Pedido, Carrinho de Compras, Pagamento

Agregados: Pedido (composto por Carrinho de Compras, Produto e Cliente), Pagamento (composto por Pedido)

Eventos: Produto Adicionado ao Carrinho, Produto Removido do Carrinho, Pedido Realizado, Pagamento Confirmado

Serviços: Processamento de Pagamento, Consulta de Pedidos, Cálculo de Descontos

Regras de Negócio: Produto não pode ser adicionado com quantidade zero, Desconto de 10% aplicado para pedidos acima de R$ 100,00, Pagamento somente é confirmado após autorização da operadora de cartão.

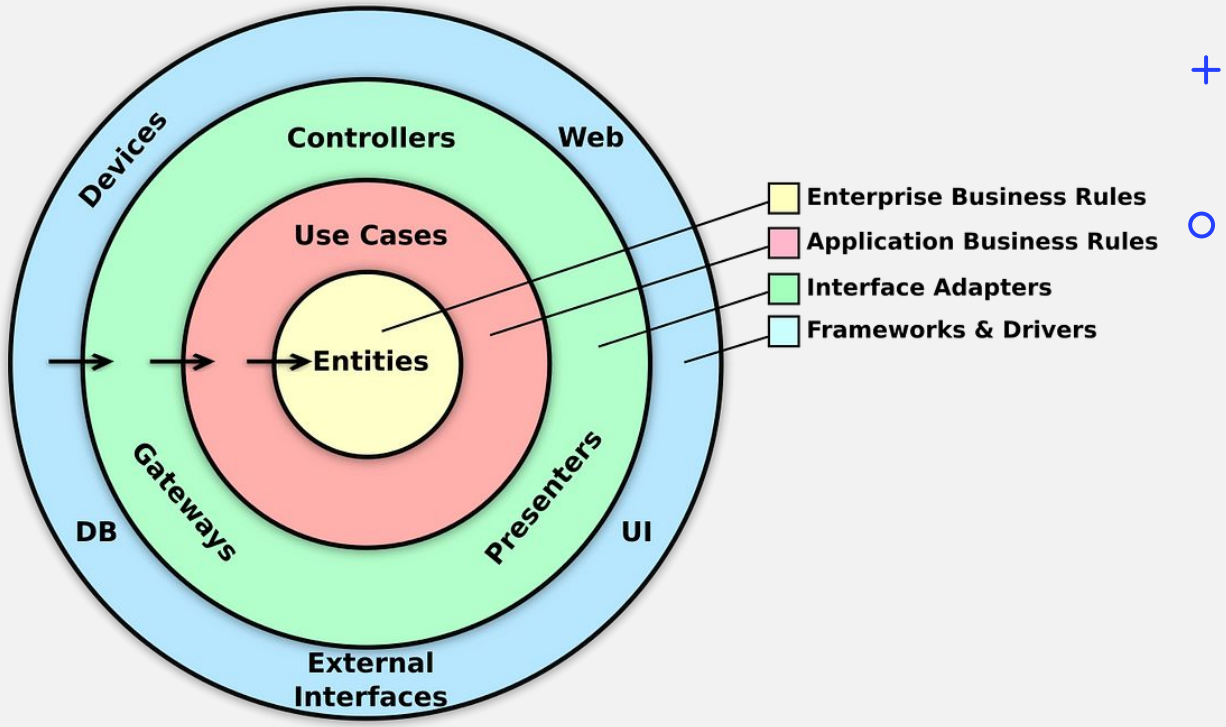


Arquitetura Limpa

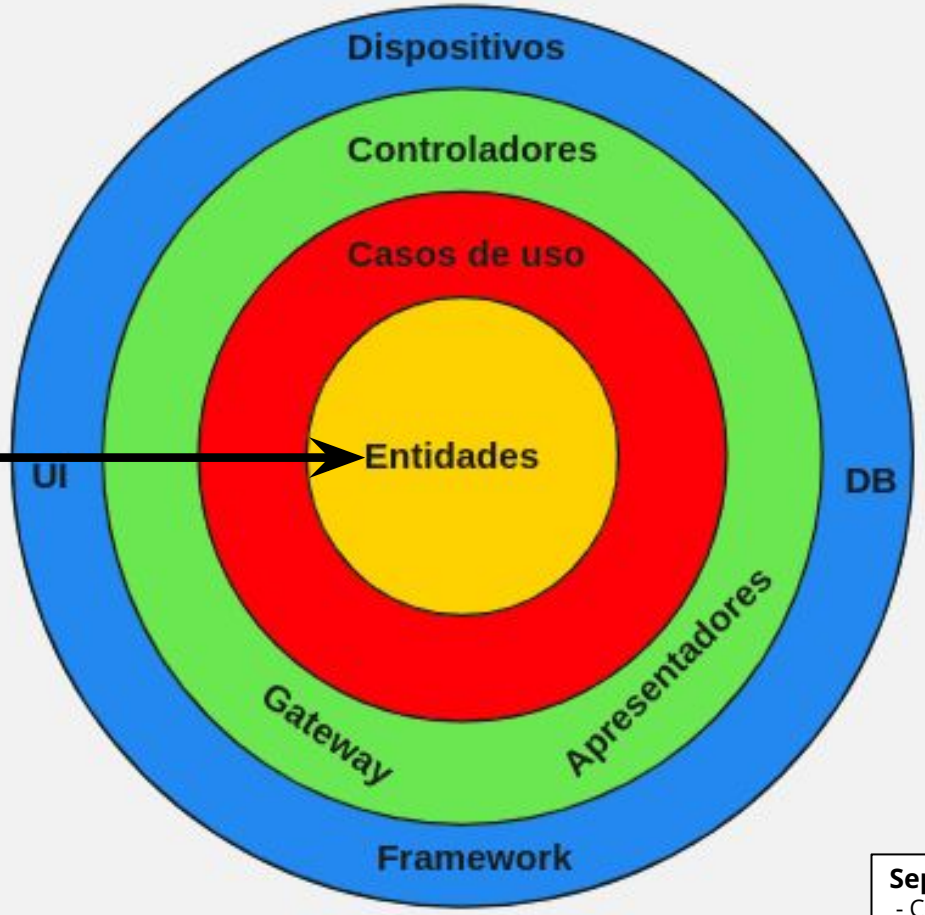
- ✔ Conceito de design proposto por Uncle Bob.

A ideia central é:

- ✔ Sistemas que sejam independentes de frameworks,
- ✔ Fáceis de testar
- ✔ Separação clara de responsabilidades.
- ✔ Busca promover um código que seja fácil de entender, manter e evoluir.



Regra de dependência



Frameworks e drivers

Aplicações externas(BD)

Adaptadores de interface

Converter dados, tratar erros e validar regras sintáticas.

Controllers – recebe requisições do usuário
Presenters – Recebe uma resposta e envia para visualização
Gateways – Chamadas para o banco de dados

Regras de negócio da aplicação

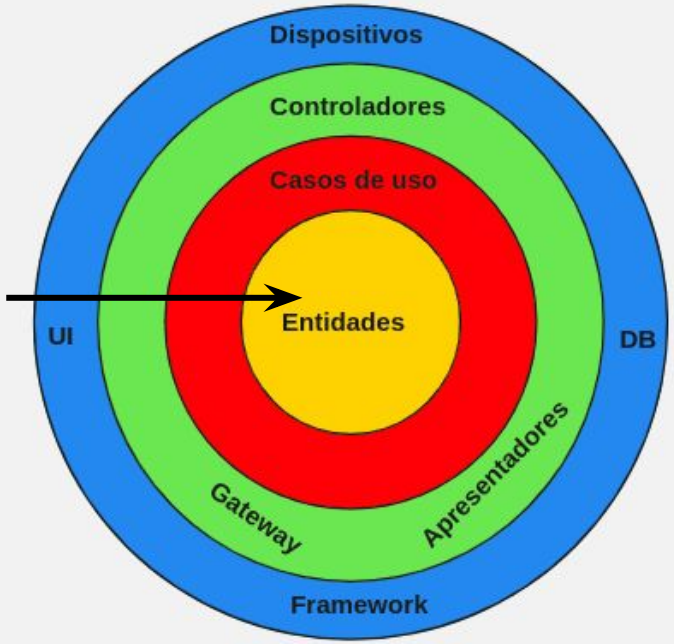
Regras de negócio da empresa

Separação de Preocupações
- Cada uma com uma responsabilidade específica.

Inversão de Dependência
- Módulos de alto nível não devem depender de módulos de baixo nível

Acoplamento Fraco
- Independência entre diferentes componentes.

Imagine uma aplicação de gerenciamento de tarefas.



Fonte: Adaptada de MARTIN (2017)

Frameworks e drivers

Implementações específicas de banco de dados, bibliotecas de terceiros.

Adaptadores de interface

Controladores REST que recebem requisições HTTP e traduzem para casos de uso.

Regras de negócio da aplicação

Criar Tarefa, Completar Tarefa.

Regras de negócio da empresa

Tarefa, Usuário.

Arquitetura Hexagonal

Conhecida como arquitetura de portas e adaptadores, é um padrão de design de software que visa tornar sistemas mais flexíveis e independentes de suas dependências externas.

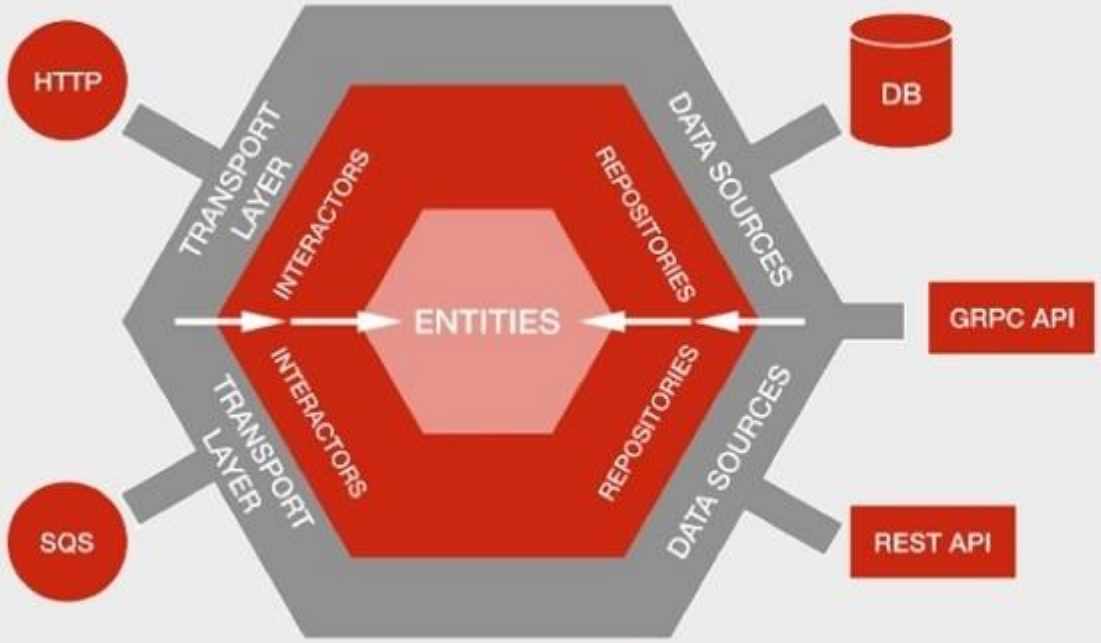
Princípios Básicos:

Isolamento da Lógica de Negócio: A lógica de negócios é isolada da infraestrutura e de outras dependências externas. Isso é feito por meio da definição de uma interface clara entre o núcleo da aplicação e o mundo exterior.

Portas e Adaptadores: A arquitetura é composta por "portas" (interfaces) e "adaptadores" (implementações concretas).

Portas são pontos de entrada para a aplicação (comoAPIs, interfaces de usuário, etc.)

Adaptadores são responsáveis por conectar essas portas com a lógica de negócios e com as tecnologias externas (bancos de dados, serviços de terceiros, etc.).



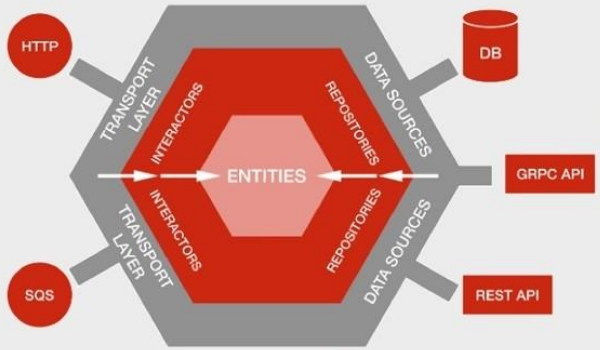
The Hexagonal Architecture

Benefícios:

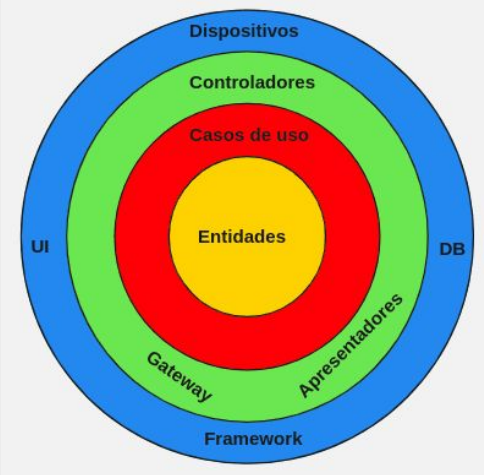
Independência de Tecnologias: Permite trocar tecnologias externas (como bancos de dados ou frameworks) com menos impacto na lógica de negócios.

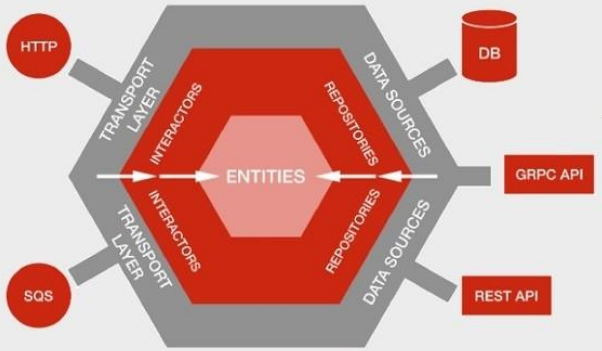
Facilidade de Testes: Facilita o teste da lógica de negócios de forma isolada, sem depender das implementações externas.

Flexibilidade: Permite uma evolução mais ágil da aplicação, pois mudanças em tecnologias externas têm menos impacto no núcleo da aplicação.



The Hexagonal Architecture





The Hexagonal Architecture

Objetivo: Garantir que a lógica de negócios e as regras de aplicação sejam independentes das implementações específicas de frameworks, bibliotecas e interfaces externas.

Objetivo: Facilitar a troca de tecnologias externas (bancos de dados, APIs, etc.) com o menor impacto possível na lógica de negócios.

Foco: Separação de Responsabilidades e Dependências

Foco: Isolamento da Lógica de Negócio

Separar as preocupações: Código deve ser dividido em partes separadas que se concentram em funções específicas e independentes.

Independência de frameworks: Código escrito de uma forma que não dependa de nenhum framework específico. Isso torna o código mais flexível e fácil de manter.

Testabilidade: Código deve ser escrito de uma forma que permita a execução de testes automatizados.

Flexibilidade: Um bom código deve ser fácil de modificar. O código deve ser escrito de uma forma que permita a fácil substituição de componentes.

SOLID

É possível aumentar a qualidade da aplicação ainda mais.

5 princípios que facilitam o processo de desenvolvimento

Single Responsibility Principle (SRP)

- **Definição:** Uma classe deve ter apenas uma responsabilidade ou propósito.
- **Benefícios**: Facilidade para fazer manutenções
- Reusabilidade das classes
- Facilidade para realizar testes
- Simplificação da legibilidade do código

Open/Closed Principle (OCP)

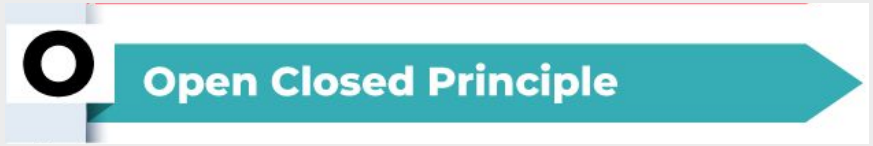
Princípio Aberto-Fechado: *"entidades de software (como classes e métodos) devem estar abertas para extensão, mas fechadas para modificação"*.
Se uma classe está aberta para modificação, quanto mais recursos adicionarmos, mais complexa ela vai ficar. O ideal é adaptar o código não para alterar a classe, mas para estendê-la. Em geral, isso é feito quando abstraímos um código para uma interface.

"Classes derivadas (ou classes-filhas) devem ser capazes de substituir suas classes-base (ou classes-mães)".
Ou seja, uma classe-filha deve ser capaz de executar tudo que sua classe-mãe faz. Esse princípio se conecta com o polimorfismo e reforça esse pilar da POO.

“Uma classe deve ter apenas uma responsabilidade ou propósito”.

Benefícios:

- Facilidade para fazer manutenções
- Reusabilidade das classes
- Facilidade para realizar testes
- Simplificação da legibilidade do código



“Entidades de software (como classes e métodos) devem estar abertas para extensão, mas fechadas para modificação”.

Se uma classe está aberta para modificação, quanto mais recursos adicionarmos, mais complexa ela vai ficar.
O ideal é adaptar o código não para alterar a classe, mas para estendê-la. Em geral, isso é feito quando abstraímos um código para uma interface.

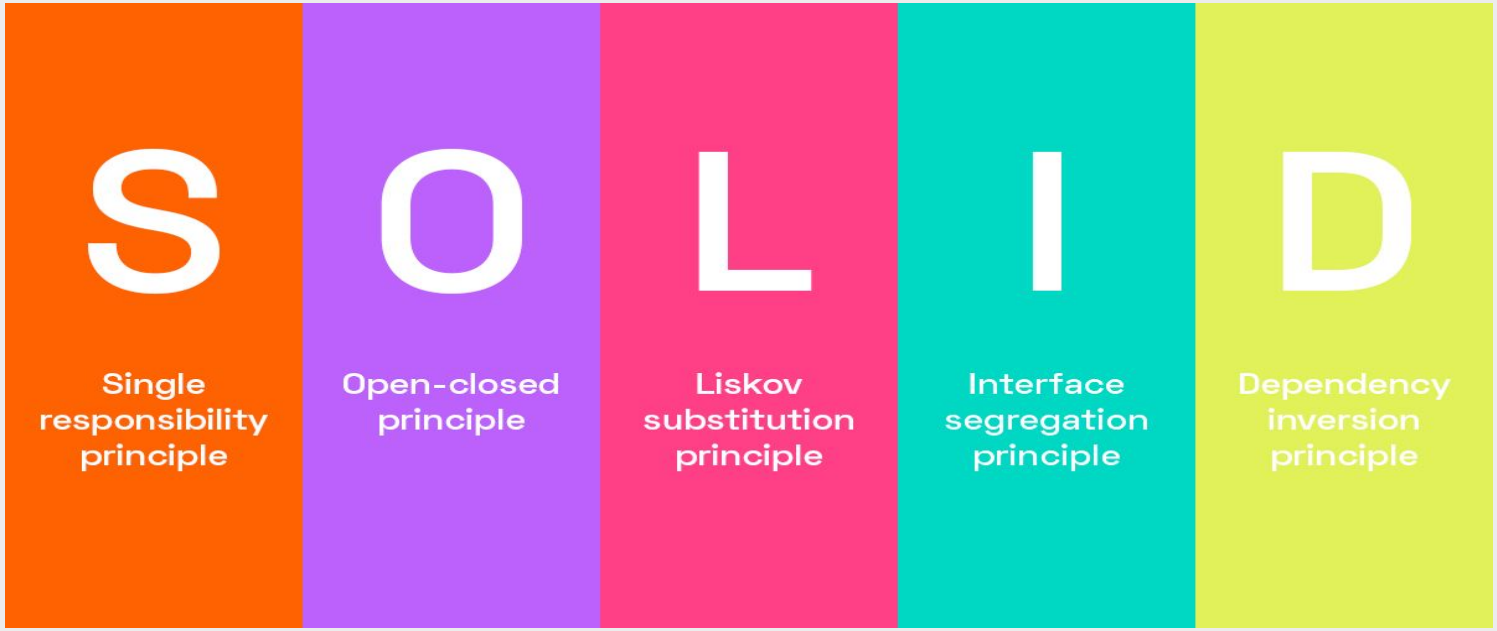
"Classes derivadas (ou classes-filhas) devem ser capazes de substituir suas classes-base (ou classes-mães)".

Ou seja, uma classe-filha deve ser capaz de executar tudo que sua classe-mãe faz. Esse princípio se conecta com o polimorfismo e reforça esse pilar da POO.

“Uma classe não deve ser forçada a implementar interfaces e métodos que não serão utilizados”.
Devemos criar interfaces específicas ao invés de termos uma única interface genérica.

“Dependa de abstrações e não de implementações concretas”.

É recomendado depender de abstrações ou interfaces que definem contratos de funcionamento. Isso promove maior flexibilidade e facilita a manutenção do sistema.



O objetivo central do SOLID é **criar sistemas que sejam fáceis de ENTENDER, MODIFICAR E ESTENDER**, permitindo que o software evolua conforme as necessidades mudam, sem sacrificar a qualidade do código ou comprometer a estrutura existente. Em suma, SOLID nasceu para **melhorar a qualidade do design de software** e garantir que os sistemas possam **CRESCER** e se **ADAPTAR** de forma ágil e sustentável.

Monolítico e Microsserviços.

Monolítico

Desenvolvidas para ser instaladas num só lugar, geralmente possuem camadas sobrepostas com responsabilidades distintas e quando é preciso realizar algum ajuste todo o sistema precisa ser atualizado no servidor de produção.

Estrutura:

Todo o código-fonte é agrupado em um único bloco, formando uma aplicação unificada. Todos os módulos (como autenticação, pagamentos, gerenciamento de usuários, etc.) estão interligados e compartilham o mesmo código e banco de dados.

Deploy:

É feito para o sistema como um todo. Se um módulo precisar de atualização, a aplicação inteira precisa ser parada e implantada novamente.

Desenvolvimento:

As equipes trabalham no mesmo código-fonte, o que pode causar dependências entre os módulos. Uma pequena mudança pode exigir a recompilação e redeploy de toda a aplicação.

Manutenção:

À medida que o sistema cresce, a manutenção pode se tornar mais difícil, pois as partes estão fortemente acopladas e uma alteração em um módulo pode impactar outros módulos.

Microsserviços

Lida com a criação de pequenos serviços autônomos. Cada microsserviço implementa uma pequena função de negócio e pode ser implantado e removido dos ambientes de produção de forma independente. Em termos técnicos, cada microsserviço pode ser escrito em uma linguagem de implementação distinta, possui o seu próprio banco de dados e se comunica com outros serviços através de chamadas REST.

Estrutura:

A aplicação é dividida em pequenos serviços independentes, onde cada serviço tem uma responsabilidade única e realiza uma função específica (por exemplo, um serviço para autenticação, outro para gerenciamento de usuários, etc.). Cada microsserviço pode ter seu próprio banco de dados.

Deploy:

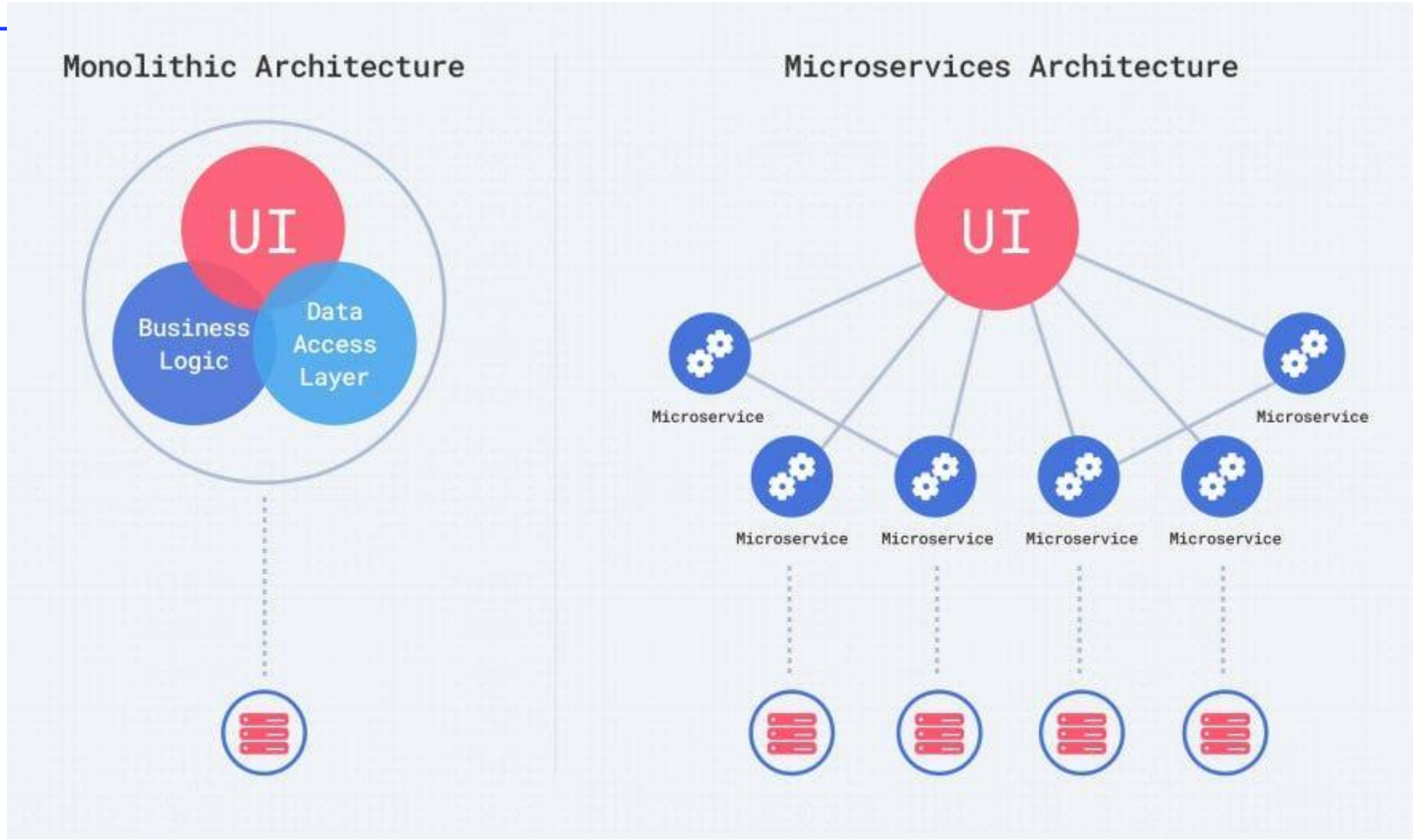
É feito por serviço. Atualizações ou novos releases de um serviço não requerem o redeploy de todo o sistema.

Desenvolvimento:

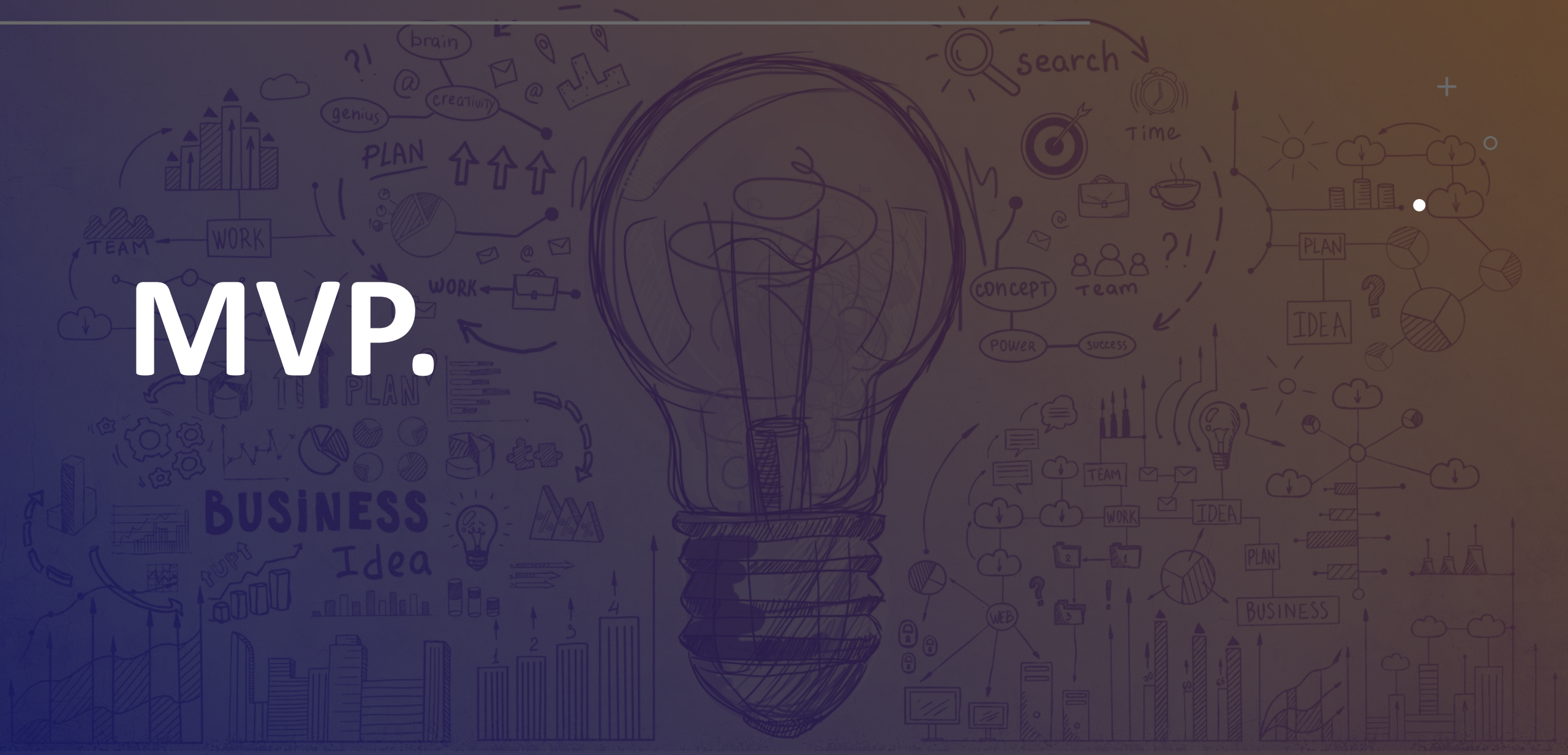
Cada microsserviço pode ser desenvolvido, testado e mantido de forma independente por diferentes equipes. Mudanças em um microsserviço não afetam diretamente outros microsserviços.

Manutenção:

Microsserviços facilitam a manutenção e atualização de sistemas grandes. Como cada serviço é independente, é possível atualizar, corrigir ou escalar um serviço sem impactar os outros.

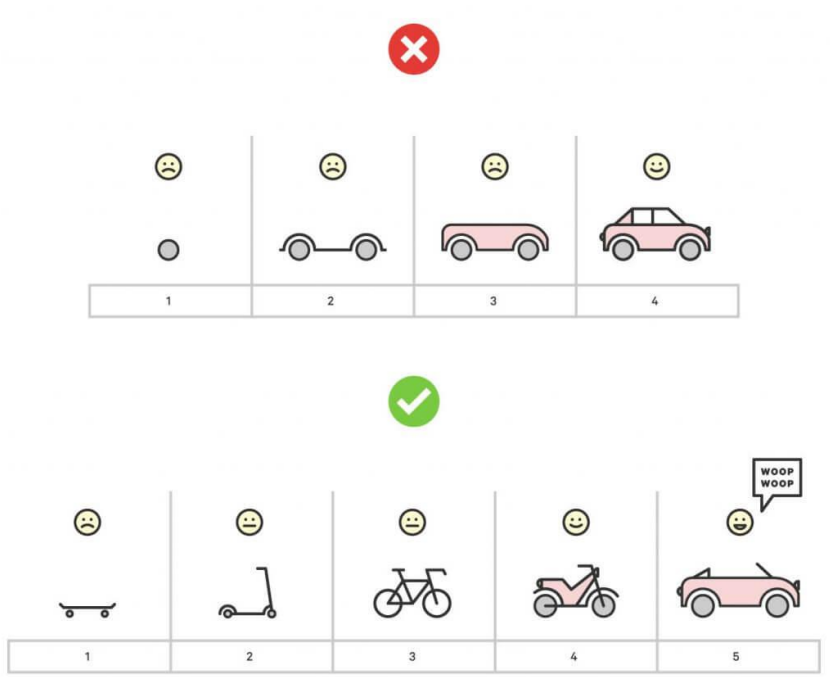

Monolithic Architecture
UI
Business Logic
Data Access Layer
Microservices Architecture
UI
Microservice
Microservice
Microservice
Microservice
Microservice
Microservice

MVP.


brain
genius
creativity
PLAN
TEAM
WORK
WORK
PLAN
BUSINESS
Idea
search
Time
concept
Team
POWER
success
PLAN
IDEA
TEAM
WORK
IDEA
PLAN
BUSINESS
WEB

MVP – Mínimo Produto Viável

É uma versão inicial de um produto com funcionalidades mínimas, suficiente para ser lançado e testado pelos usuários reais, permitindo validar e obter feedback.



MVP – Mínimo Produto Viável

- ✔ Atenção aos requisito e entregar o essencial
- ✔ Evolução através de feedbacks (muitas vezes o usuário nem sabe o que realmente quer)
- ✔ Realizar várias entregas para o cliente (entrega contínua)
- ✔ Rapidez na validação em mercado – O que poupa tempo e investimento
- ✔ Pequenas entregas testáveis antes de disponibilizar em massa.
- ✔ Melhor definição das prioridade e foco no que realmente importa.

Obrigada!


brain
genius
creativity
PLAN
TEAM
WORK
WORK
search
Time
concept
Team
POWER
success
PLAN
IDEA
BUSINESS
Idea
TEAM
WORK
IDEA
PLAN
WEB
BUSINESS